A simplified algorithm computing all s-t bridges and articulation points

Cairo, Massimo

2021-12-31

Cairo , M , Khan , S , Rizzi , R , Schmidt , S S , Tomescu , A & Zirondelli , E C 2021 , ' A simplified algorithm computing all s-t bridges and articulation points ' , Discrete Applied Mathematics , vol. 305 , pp. 103-108 . https://doi.org/10.1016/j.dam.2021.08.026

http://hdl.handle.net/10138/334908
https://doi.org/10.1016/j.dam.2021.08.026

Discrete Applied Mathematics 305 (2021) 103–108

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Note

A simplified algorithm computing all *s*-*t* bridges and articulation points

Massimo Cairo [a], Shahbaz Khan [a], Romeo Rizzi [b], Sebastian Schmidt [a], Alexandru I. Tomescu [a,*], Elia C. Zirondelli [c]

[a] *Department of Computer Science, University of Helsinki, Finland*
[b] *Department of Computer Science, University of Verona, Italy*
[c] *Department of Mathematics, University of Trento, Italy*

ARTICLE INFO

Article history:
Received 6 January 2021
Received in revised form 11 June 2021
Accepted 22 August 2021
Available online xxxx

Keywords:
Reachability
Graph algorithm
Strong bridge
Strong articulation point
Directed graph

ABSTRACT

Given a directed graph G and a pair of nodes s and t, an *s-t bridge* of G is an edge whose removal breaks all s-t paths of G. Similarly, an *s-t articulation point* of G is a node whose removal breaks all s-t paths of G. Computing the sequence of all s-t bridges of G (as well as the s-t articulation points) is a basic graph problem, solvable in linear time using the classical min-cut algorithm (Ford and Fulkerson, 1956).

We show a simplified and self-contained algorithm computing all s-t bridges and s-t articulation points of G, based on a single graph traversal from s to t avoiding an arbitrary s-t path, which is interrupted at the s-t bridges. Its proof of correctness uses simple inductive arguments, making the problem an application of merely *graph traversal*, rather than of the more complex *maximum flow* problem.

1. Introduction

Connectivity and reachability are fundamental graph-theoretical problems studied extensively in the literature [5,6,9,11]. A key notion underlying such algorithms is that of edges (or nodes) critical for connectivity or reachability. The most basic variant of these is a bridge (or articulation point), which is defined as follows. A *bridge* of an undirected graph, also referred as a *cut edge*, is an edge whose removal increases the number of connected components. Similarly, a *strong bridge* in a (directed) graph is an edge whose removal increases the number of strongly connected components of the graph. (Strong) articulation points are defined in an analogous manner by replacing the edge with a node.

Special applications consider the notion of bridges to be parameterised by the nodes that become disconnected upon their removal [10,13]. Given a node s, we say that an edge is an *s bridge* (also referred as *edge dominators* from source s [10]) if there exists a node t that is no longer reachable from s when the edge is removed. Moreover, given both nodes s and t, an *s-t bridge* (or s-t articulation point) is an edge (or node) whose removal makes t no longer reachable from s.

For undirected graphs, the classical algorithm by Tarjan [12] computes all bridges and articulation points in linear time. However, for directed graphs only recently Italiano et al. [10] presented an algorithm to compute all strong bridges and strong articulation points in linear time. They also showed that classical algorithms [8,13] compute s bridges in linear time. The s articulation points (or *dominators*) are extensively studied resulting in several linear-time algorithms [1–3].

* Corresponding author.
E-mail addresses: shahbaz.khan@helsinki.fi (S. Khan), romeo.rizzi@univr.it (R. Rizzi), sebastian.schmidt@helsinki.fi (S. Schmidt), alexandru.tomescu@helsinki.fi (A.I. Tomescu), eliacarlo.zirondelli@unitn.it (E.C. Zirondelli).

https://doi.org/10.1016/j.dam.2021.08.026

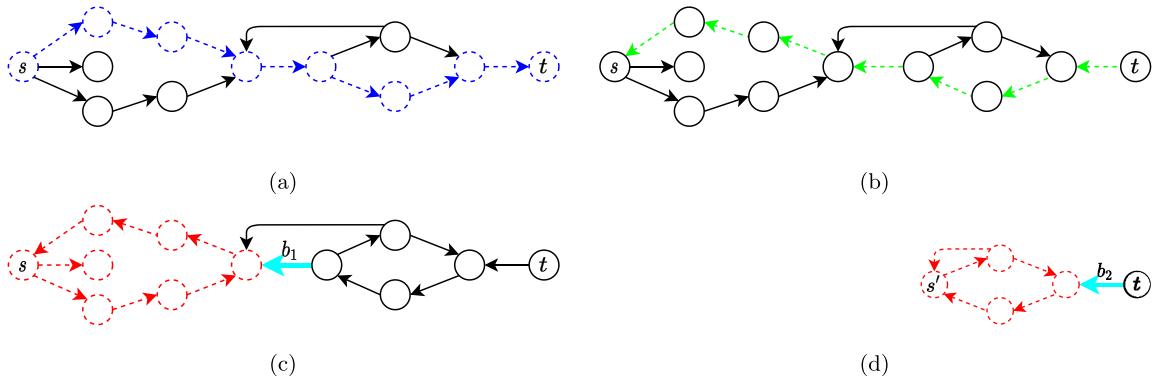

Fig. 1. Example for Ford–Fulkerson based computation of *s*-*t* bridges. (a) The path of unit flow is computed (blue, dashed). (b) The edges of the path are reversed (green, dashed) in the residual graph. (c) The reachable nodes from s are identified as the first minimal *s*-*t* cut (red, dashed), and the reversed path-edge entering the cut as the *s*-*t* bridge (cyan, bold). (d) The next cut and *s*-*t* bridge is computed similarly by contracting the first cut and *s*-*t* bridge as s'.

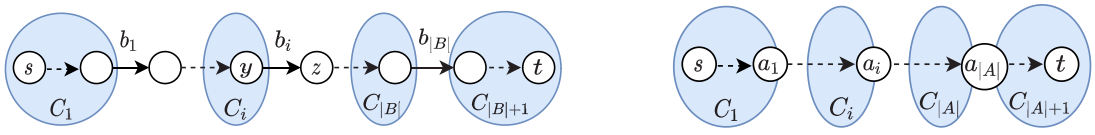

Fig. 2. The bridge sequence $B = \{b_1, \ldots, b_{|B|}\}$ and the articulation sequence $A = \{a_1, \ldots, a_{|A|}\}$ along an *s*-*t* path, with their corresponding components $\mathcal{C} = \{C_1, \ldots, C_k\}$, where $k = |B| + 1$ or $|A| + 1$, respectively.

The *s*-*t* bridges were essentially studied as minimum *s*-*t* cuts in network flow graphs, since an *s*-*t* bridge is a cut of *unit* size. The classical Ford Fulkerson algorithm [7] can be used to identify the first *s*-*t* bridge in the residual graph after pushing *unit* flow in the network. For this, one computes a cut as the reachable nodes from s in the residual network (with the unit flow-edges reversed). The reversed edge entering the cut is then the first *s*-*t* bridge. By contracting the entire cut to s, one can continue finding the next *s*-*t* bridge and so on. See Fig. 1 for an example. Since *s*-*t* bridges limit the maximum flow to *one*, the algorithm completes in linear time. The components formed by such contracted cuts are also useful for an application [4] in Bioinformatics. Hence finding all *s*-*t* bridges was understood as an application of maximum flow.

We show that computing all *s*-*t* bridges can be simplified to performing *a single* graph traversal from s to t, avoiding an arbitrary *s*-*t* path, and which is *interrupted* at the *s*-*t* bridges. While this algorithm is inspired by the above network flow approach, it is a simple application of graph traversal, and it has a simpler proof of correctness based on induction, making it independent of the theory of network flows. We first present the algorithm for *s*-*t* bridges, and then extend it to *s*-*t* articulation points.

2. Preliminaries

Let $G := (V, E)$ be a fixed directed graph, where V is a set of n nodes and E a set of m edges, with two given nodes $s, t \in V$. Let $G \setminus X$ denote the resulting graph after removing all edges in X from G. Given an edge $e = (u, v)$, $\textsc{head}(e) = v$ denotes its *head* and $\textsc{tail}(e) = u$ denotes its *tail*.

Let $B = \{b_1, b_2, \ldots, b_{|B|}\}$ be the set of *s*-*t* bridges of G. By definition, for all $b_i \in B$ there exists no path from s to t in $G \setminus b_i$ (see Fig. 2), and all *s*-*t* bridges in B appear on every *s*-*t* path in G. Further, the *s*-*t* bridges in B are visited in the same order by every *s*-*t* path in G, which follows by considering the reachability in $G \setminus b_i$ for any bridge b_i. Thus, abusing the notation we define B to be a *sequence of s-t bridges* ordered by their visit time on any *s*-*t* path.

Also, such a bridge sequence B implies an increasing part of the graph being reachable from s in $G \setminus b_i$, as i increases. We thus divide the graph reachable from s into *bridge components* $\mathcal{C} = \{C_1, C_2, \ldots, C_{|B|+1}\}$, where C_i (for $i \leq |B|$) denotes the part of graph that is reachable from s in $G \setminus b_i$ but was not reachable in $G \setminus b_{i-1}$ (if any). Additionally, for notational convenience we assume $C_{|B|+1}$ to be the part of the graph reachable from s in G, but not in $G \setminus b_{|B|}$ (see Fig. 2). Since bridge components are separated by *s*-*t* bridges, every *s*-*t* path enters C_i at a unique vertex ($\textsc{head}(b_{i-1})$ or s for C_1) referred as its *entry*, and it leaves C_i at a unique vertex ($\textsc{tail}(b_i)$ or t for $C_{|B|+1}$) referred as its *exit*.

The *s*-*t* *articulation points* are similarly defined as the set of nodes $A = \{a_1, a_2, \ldots, a_{|A|}\}$, such that removal of any $a_i \in A$ disconnects all *s*-*t* paths in G. They also follow a fixed order in every *s*-*t* path defining A as a sequence with the corresponding components $\mathcal{C}$ (see Fig. 2). The *entry* and *exit* of an articulation component C_i are the preceding and succeeding *s*-*t* articulation points (if any), else s and t respectively.

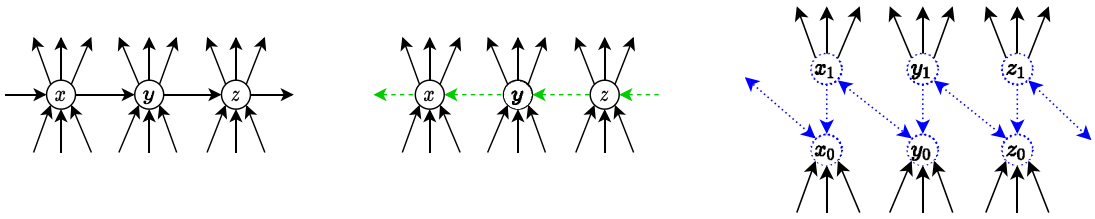

Fig. 3. Transformation of the graph along an s-t path for computing the s-t bridges (shown in green dashed) and the s-t articulation points (shown in blue dotted).

3. Algorithm

Our algorithm can essentially be described as a *forward search* from s to t avoiding an arbitrary s-t path, which is *interrupted* at the s-t bridges (or s-t articulation points). It discovers the bridge sequence B (or articulation sequence A) in order as the search proceeds. Our algorithm requires graph transformations to simplify the computation of s-t bridges and s-t articulation points. In fact the second transformation reduces the problem of computing s-t articulation points to that of computing s-t bridges.

The algorithm first chooses an arbitrary s-t path P in G. Then it performs a forward search from s to reach t avoiding the edges of P. The aim is to transform the graph such that this search is interrupted by exactly the s-t bridges, since all the s-t bridges lie on P (by definition). Thus, if the forward search stops before reaching t, it necessarily requires to traverse an edge b_1 in P (i.e. the first s-t bridge) to reach t. So we continue to forward search from $\textsc{head}(b_1)$ until we stop again to find the next s-t bridge, and so on until we reach t. When the search is interrupted for the ith time, we look at the last node y on P, that was visited by the search. The s-t bridge b_i is then identified as the outgoing edge of y on P.

Graph Transformation. Now, to perform the transformation one approach can be to simply remove the edges of P. However, merely avoiding edges of P does not necessitate that the search is interrupted only at the s-t bridges. This is because when this search reaches a node y on P, it clearly implies that all the edges preceding y on P are not s-t bridges. Further, a node (say x) preceding y on P may be used to reach further along P beyond y. Thus, all such nodes preceding y must be traversed before assuming that the search is interrupted at an s-t bridge. This extra procedure of making such nodes reachable from y for the forward search can be embedded in the original forward search using a modified transformation, where we reverse the path P instead of removing it (see Fig. 3). Thus, on reaching y every node preceding it is reachable by the forward search, ensuring that it is interrupted only at s-t bridges. Note that this transformation resembles the residual graph after a *unit flow* is pushed through the network.

For computing s-t articulation points, we require this search to be interrupted on s-t articulation points instead of s-t bridges. So each node x on P is split into two nodes x_0 and x_1, having all incoming edges now incoming to x_0 and all outgoing edges now outgoing from x_1. Further, we have the *internal edge* of the node x from x_0 to x_1. This transformation maintains the path P where each node is split into two by an internal edge, where the internal edge of a node x is an s-t bridge if and only if x is an s-t articulation point. Now, when the previous transformation is applied, it reverses the s-t path P thereby reversing the internal edges (x_0, x_1) as well. Finally, to prevent the search to interrupt at the original s-t bridges, the non-internal edges of P are added back to G (see Fig. 3). Note that this adds the original edges of P in both directions.

We now formally describe the algorithm (refer to the pseudocode in Algorithm 1). After choosing an arbitrary s-t path P, it transforms the graph as described above, by reversing the path P. Along with computing the bridge sequence B and bridge components $\mathcal{C}$, we also ensure access to the component of a node v. This is stored in $comp[v]$ which is initialised to 0, also serving as an indicator that v is not visited. Thereafter, it initiates the search with a queue Q containing s, the entry of C_1. The forward search continues removing and visiting nodes in Q, and adding their unvisited out-neighbours back to Q, until Q becomes empty and the search stops. Every node v visited during this search is assigned to C_i and has $comp[v] = i$. If t is not visited yet, the last node y in P (closest to t), that was visited during the search is identified (i.e. the *exit* of C_i), by looking at the nodes in P starting from *entry* of C_i. The s-t bridge b_i is identified to be the outgoing edge of y in P (say (y, z), see Fig. 2), and added to B. The forward search is then continued from z (i.e. the *entry* of C_{i+1}) by adding it to Q, and so on. Otherwise, if t was already visited when the search stopped, we terminate the algorithm with bridge sequence B and the bridge components and node associations in $\mathcal{C}$ and $comp[\cdot]$ respectively. For an example execution of the algorithm, see Fig. 4.

4. Analysis and Correctness

The algorithm essentially performs three procedures that need to be analysed. *Firstly,* a standard BFS traversal requiring $O(m + n)$ time, for computing an s-t path P, and for the interrupted forward search which only visits the previously unvisited nodes (using the Queue Q and the inner loop). *Secondly,* traversing through the path P in $O(n)$ time, for

Algorithm 1: Bridge sequence and Bridge components

Input: Graph $G := (V, E)$, $s, t \in V$
Output: Bridge sequence B and bridge components $\mathcal{C}$ and associations $comp[\cdot]$

```
1  P ← Arbitrary s-t path in G
2  G ← (G \ P) ∪ P^{-1}                          // graph transform, reverse P
3  i ← 1
4  while comp[t] = 0 do
5      if i = 1 then Add s to Q and C_1, comp[s] ← 1      // initialise search from s
6      else
7          y ← Last node on P with comp[u] ≠ 0            // traverse P from entry of C_i
8          z ← Successor of y in P                        // where b_{i-1} : (y, z) ∈ P
9          Add (y, z) to B
10         Add z to Q and C_i, comp[z] ← i                // continue search from z
11     while Q ≠ ∅ do                                     // forward search
12         u ← Remove node from Q
13         forall the (u, v) ∈ E where comp[v] = 0 do
14             Add v to Q and C_i, comp[v] ← i
15     i ← i + 1
```

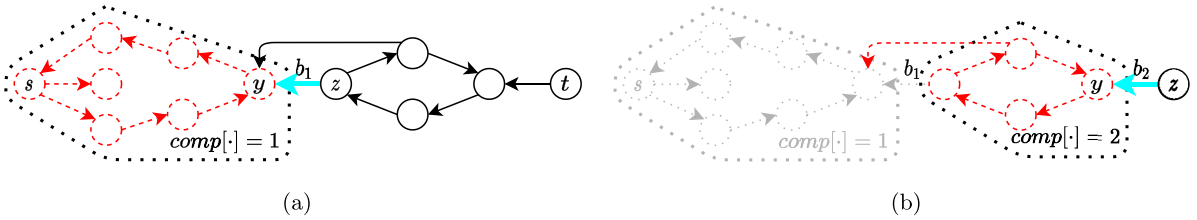

Fig. 4. Example execution of Algorithm 1. We use the same s-t path as in Fig. 1 (a), and transform the graph as in Fig. 1 (b). (a) We compute the reachable subgraph from s (red, dashed), the last node y on P reachable from s and its successor z in P. Thus, (y, z) is identified as the first s-t bridge (cyan, bold), and the reachable nodes as C_1 (circled with a dotted line). (b) We continue the traversal from the former z, ignoring already reached parts (grey, dotted), to identify the next s-t bridge and C_i as before.

transforming the graph by reversing $|P|$ edges, and for identifying the last visited node on P which traverses P once during the entire algorithm. *Thirdly,* updating B, $\mathcal{C}$ and $comp[t]$ requires to visit the outer loop for each s-t bridge, requiring total $O(n)$ time as the number of s-t bridges and nodes are $O(n)$. Also, the transform for s-t articulation points requires $O(m)$ time for splitting the nodes and adding the corresponding edges. Thus, the algorithm requires overall $O(m + n)$ time to compute all the s-t bridges and the associated components.

The correctness of the algorithm can be proven by maintaining the following invariant.

Invariant $\mathcal{I}$: *In the transformed graph, the forward search started from the entry of* C_i

(a) *visits exactly the nodes in* C_i, *and*
(b) *visits the nodes in* P *only up to the exit of* C_i.

Proof. The graph transformation only affects the paths passing through the edges of P, as the remaining edges are unaffected. Hence, to prove $\mathcal{I}(a)$ it is sufficient to prove that the nodes on P within C_i, say v_1 (entry), $v_2, \ldots, v_k$(exit), are reachable from the entry v_1. We prove it by induction over the nodes v_j, where the base case ($j = 1$) is trivially true.

For any v_j (see Fig. 5), assuming nodes up to v_{j-1} are reachable from the entry, we consider the edge $e_j = (v_{j-1}, v_j)$. Since e_j is not an s-t bridge, there exists a path P' from s to t (and hence from entry v_1 to exit v_k) without using e_j in the original graph. Let u and v respectively be the nodes at which P' leaves P for the last time before e_j, and the node at which P' joins P again after e_j. The nodes u and v necessarily exist as the path P' passes through v_1 and v_k. Note that the subpath of P' from u to v does not pass through any edge in P by definition. Now, u ($= v'_{j'}, j' < j$) is reachable from v_1 in the transformed graph by induction hypothesis, and there is a path from v to v_j in the reverse path of P. Thus, v_j is reachable from u (and hence v_1) through P' in the transformed graph.

Using induction, every v_j and hence the entire C_i is reachable from v_1, proving $\mathcal{I}(a)$. Further, since b_i is an s-t bridge which is reversed and hence removed in the transformed graph, the forward search cannot reach HEAD(b_i). This is because

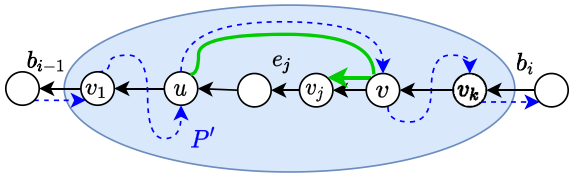

Fig. 5. Reachability of v_j from v_1 (entry) in the transformed graph. The path P' (blue, dashed) is the alternate s-t path avoiding e_j, which leaves P before e_j for the last time at u, then joins P again after e_j at v. The subpath of P' from u to v followed by reverse path of P to v_j (green, bold), reaches v_j from u and hence v_1.

Algorithm 2: Articulation sequence and its components

Input: Graph $G := (V, E)$, $s, t \in V$
Output: Articulation sequence A and its components $\mathcal{C}$ and associations $comp[\cdot]$

$P \leftarrow$ Arbitrary s-t path in G
$G \leftarrow G \cup P^{-1}$ // graph transform, reverse P
forall the $x \in V$ **do**
  if $x \in P \setminus \{s, t\}$ **then** $out[x] \leftarrow 0$ // out edges of x except on P^{-1} not usable
  else $out[x] \leftarrow 1$ // all out edges of x usable
$i \leftarrow 1$
while $comp[t] = 0$ **do**
  if $i = 1$ **then** Add s to Q and C_1, $comp[s] \leftarrow 1$ // initialise search from s
  else
    $y \leftarrow$ Last node on P with $comp[u] \neq 0$ // traverse P from entry of C_i
    Add y to A, $out[y] \leftarrow 1$ // continue search with all out edges of y
    Add y to Q and C_i, $comp[y] \leftarrow i$ // continue search from y
  while $Q \neq \emptyset$ **do** // forward search
    $u \leftarrow$ Remove node from Q
    forall the $(u, v) \in E$ ***if*** $out[u] = 1$ ***else*** $(u, v) \in P^{-1}$ **do** // usable out edges
      if $comp[v] = 0$ ***or*** $((u, v) \in P^{-1}$ ***and*** $out[v] = 0)$ **then**
        Add v to Q and C_i, $comp[v] \leftarrow i$
        if $(v, u) \in P$ **then** $out[v] = 1$ // since u is visited, $v \notin A$
  $i \leftarrow i + 1$

there is no other edge from C_i to C_{i+1} including the reversed edges of P. Hence, the last node of P (closest to t) visited by the forward search is exactly the exit v_k proving $\mathcal{I}(b)$. By induction, assuming C_{i-1} is computed correctly, all C_i would be computed correctly proving $\mathcal{I}$ for all C_i. □

Articulation points can be computed by Algorithm 1 using the transformation described in Fig. 3. However, for the sake of completeness we also present the pseudocode (Algorithm 2) for computing them directly without this transformation. The equivalence of visiting v_1 in Fig. 3 is having $out[v] = 1$, implying that all out edges of v are usable by the forward search. The analysis and correctness follow the same arguments. Thus, we have the following theorem.

Theorem 1. *Given a graph $G := (V, E)$ with n nodes, m edges and $s, t \in V$, there exists an algorithm to compute all s-t bridges (or articulation points) along with their component associations, in $O(m + n)$ time.*

5. Conclusions

We have simplified the maximum flow-based algorithm for computing s-t bridges (or articulation points) along with their component associations, obtaining an algorithm based on a single graph traversal. The resulting algorithm has a simple inductive proof, independent of the theory of flows, and it is potentially easier to use in practice compared to flow-based approaches.

Acknowledgements

This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO), and by the Academy of Finland (grants No. 322595, 328877).

References

[1] S. Alstrup, D. Harel, P.W. Lauridsen, M. Thorup, Dominators in Linear Time, SIAM J. Comput. 28 (6) (1999) 2117–2132.
[2] A.L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R.E. Tarjan, J.R. Westbrook, Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems, SIAM J. Comput. 38 (4) (2008) 1533–1573.
[3] A.L. Buchsbaum, H. Kaplan, A. Rogers, J.R. Westbrook, Corrigendum: a new, simpler linear-time dominators algorithm, ACM Trans. Program. Lang. Syst. 27 (3) (2005) 383–387.
[4] M. Cairo, S. Khan, R. Rizzi, S.S. Schmidt, A.I. Tomescu, Safety in s-t Paths, Trails and Walks, 2020, CoRR, arXiv:2007.04726.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, Third ed., MIT Press, 2009.
[6] R. Diestel, Graph Theory, Fourth, Graduate Texts in Mathematics, vol. 173, Springer, 2010.
[7] L.R. Ford, D.R. Fulkerson, Maximal Flow Through a Network, Canad. J. Math. 8 (1956) 399–404, http://dx.doi.org/10.4153/CJM-1956-045-5.
[8] H.N. Gabow, R.E. Tarjan, A Linear-Time Algorithm for a Special Case of Disjoint Set Union, J. Comput. System Sci. 30 (2) (1985) 209–221.
[9] J.L. Gross, J. Yellen, P. Zhang, Handbook of Graph Theory, Second Edition, second ed., Chapman & Hall/CRC, 2013.
[10] G.F. Italiano, L. Laura, F. Santaroni, Finding strong bridges and strong articulation points in linear time, Theoret. Comput. Sci. 447 (2012) 74–84.
[11] S.S. Skiena, The Algorithm Design Manual, second ed., Springer Publishing Company, Incorporated, 2008.
[12] R.E. Tarjan, A Note on Finding the Bridges of a Graph, Inf. Process. Lett. 2 (6) (1974) 160–161.
[13] R.E. Tarjan, Edge-Disjoint Spanning Trees and Depth-First Search, Acta Inf. 6 (1976) 171–185.